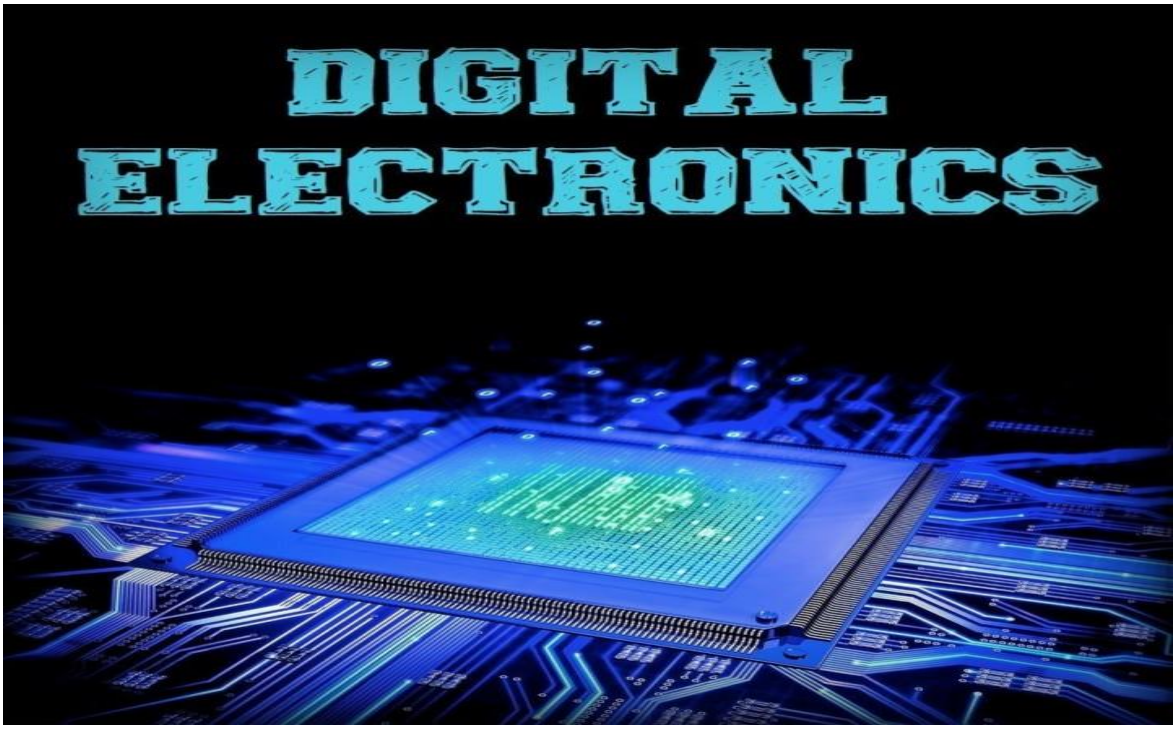
Abanob Evram

Assignmen3



[Q1]

1) Implement the following latch as specified below

Parameters

LAT_WIDTH: Determine the width of input data and output q

Ports

Name	Type	Description
aset	Input	Asynchronous set input. Sets q[] output to 1.
data[]		Data Input to the D-type latch with width LAT_WIDTH
gate		Latch enable input
aclr		Asynchronous clear input. Sets q[] output to 0.
q[]	Output	Data output from the latch with with LAT_WIDTH

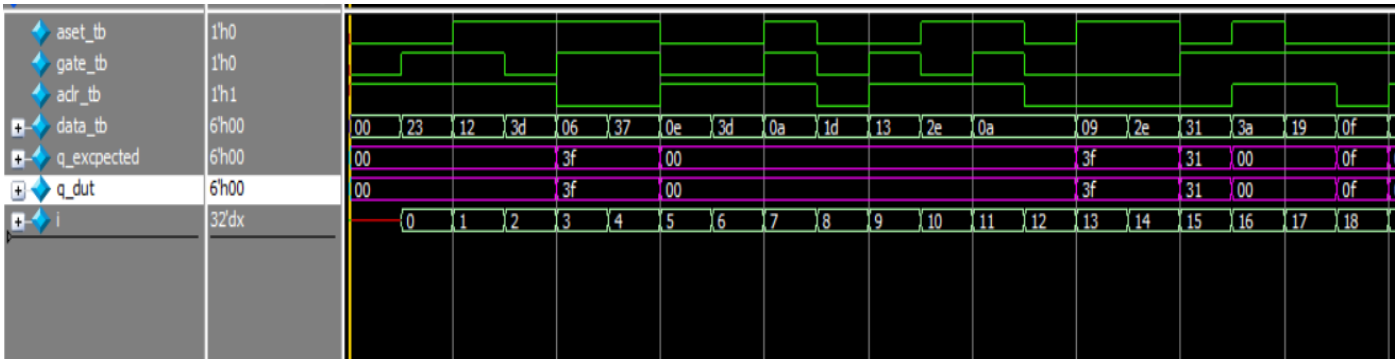
The design code:

```
module latch(aset,data,gate,aclr,q);
 parameter LAT_WIDTH=6;
 input aset,gate,aclr;
 input [LAT_WIDTH-1:0] data;
 output reg [LAT_WIDTH-1:0] q;
  always @(*) begin
    if(aclr)
     q<=0;
   else if (aset)
     q<={LAT_WIDTH{1'b1}};
   else if (gate)
     q<=data;
  end
endmodule
```

The testbench code:

```
module latch_tb();
 parameter LATCH_WIDTH_tb = 6;
 reg aset_tb,gate_tb,aclr_tb;
 reg [LATCH_WIDTH_tb-1:0] data_tb,q_excpected;
 wire [LATCH_WIDTH_tb-1:0] q_dut;
 latch #(LATCH_WIDTH_tb) dut(aset_tb,data_tb,gate_tb,aclr_tb,q_dut);
 integer i;
  initial begin
    aset_tb=0;aclr_tb=1;data_tb=0;gate_tb=0;q_excpected=0;
    if(q_dut!=q_excpected) begin
      $display("errorr");
      $stop;
      end
    #10
    for (i=0;i<99;i=i+1) begin
      aset_tb=$random;
      aclr_tb=$random;
      gate_tb=$random;
      data_tb=$random;
```

```
        if(aclr_tb)
          q_excpected<=0;
        else if (aset_tb)
          q_excpected<={LATCH_WIDTH_tb{1'b1}};
        else if (gate_tb)
          q_excpected<=data_tb;
        #10
        if(q_excpected!=q_dut)begin
                $display("Errrror");
                $stop;
        end
      end
        $stop;
    end
endmodule
```



[Q2]

```
module reg_blocking1(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        q1 = 0;
        q2 = 0;
        out = 0;
    end
    else begin
        q1 = in;
        q2 = q1;
        out = q2;
    end
end

endmodule
```

```
module reg_non_blocking1(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        q1 <= 0;
        q2 <= 0;
        out <= 0;
    end
    else begin
        q1 <= in;
        q2 <= q1;
        out <= q2;
    end
end

endmodule
```

```
module reg_blocking2(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        q2 = 0;
        q1 = 0;
        out = 0;
    end
    else begin
        q2 = q1;
        q1 = in;
        out = q2;
    end
end

endmodule
```

```
module reg_non_blocking2(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        q2 <= 0;
        q1 <= 0;
        out <= 0;
    end
    else begin
        q2 <= q1;
        q1 <= in;
        out <= q2;
    end
end

endmodule
```

```
module reg_blocking3(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        out = 0;
        q2 = 0;
        q1 = 0;
    end
    else begin
        out = q2;
        q2 = q1;
        q1 = in;
    end
end

endmodule
```

```
module reg_non_blocking3(clk, rst, in, out);
input in, rst, clk;
output reg out;

reg q1, q2;

always @(posedge clk) begin
    if (rst) begin
        out <= 0;
        q2 <= 0;
        q1 <= 0;
    end
    else begin
        out <= q2;
        q2 <= q1;
        q1 <= in;
    end
end

endmodule
```

2-

```
module comb_blocking1(clk, a, b, c, y);
input a, b, c, clk;
output reg y;

reg x;

always @(posedge clk) begin
  x = a & b;
  y = x | c;
end

endmodule
```

```
module comb_non_blocking1(clk, a, b, c, y);
input a, b, c, clk;
output reg y;

reg x;

always @(posedge clk) begin
  x <= a & b;
  y <= x | c;
end

endmodule
```

```
module comb_blocking2(clk, a, b, c, y);
input a, b, c, clk;
output reg y;

reg x;

always @(posedge clk) begin
  y = x | c;
  x = a & b;
end

endmodule
```

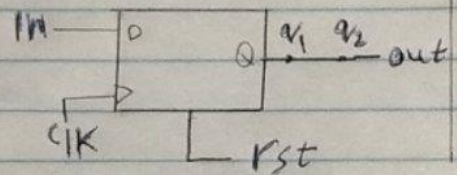
```
module comb_non_blocking2(clk, a, b, c, y);
input a, b, c, clk;
output reg y;

reg x;

always @(posedge clk) begin
  y <= x | c;
  x <= a & b;
end

endmodule
```

```
module comb_non_blocking3(clk, a, b, c, y);
input a, b, c, clk;
output reg y;

reg x;
reg y_comb;

always @(*) begin
  x = a & b;
  y_comb = x | c;
end

always @(posedge clk) begin
  y <= y_comb;
end

endmodule
```

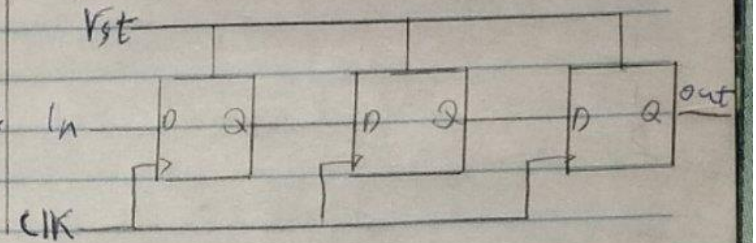
Page: Date:

[2]

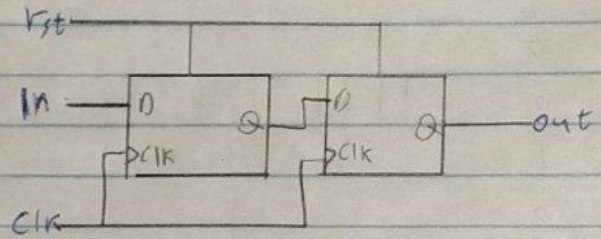
1- reg_blocking1



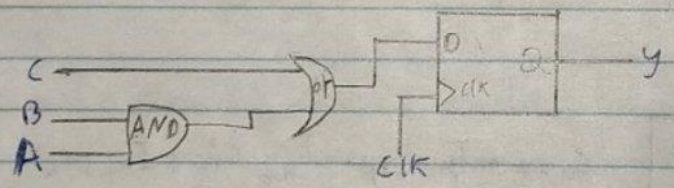
reg_nonblocking1 // reg_non_blocking2
reg_nonblocking3 // reg_blocking3



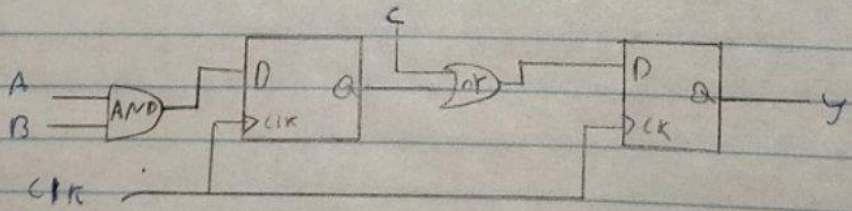
2- reg-blocking2



2- comb_blocking1 // comb-non_blocking2



Comb_non-blocking1 // comb-blocking2 // comb_non_blocking2



3)

1. Implement 4-bit counter with asynchronous active low set using behavioral modelling that increment from 0 to 15
   - Input:
     - clk
     - set (sets all bits to 1)
     - out (4-bits)
2. Use the structural counter done in Assignment3_v13 as a golden reference.
3. Test the above behavioral design using a self-checking testbench
   - Testbench should instantiate both structural and behavioral counters.

[Q3]

The design code:

```
module Counter(clk,set,out);
 input clk,set;
 output reg [3:0] out;
   always @(posedge clk or negedge set) begin
      if(~set)
         out<=4'b1111;
      else
         out<=out+1;
   end
endmodule
```

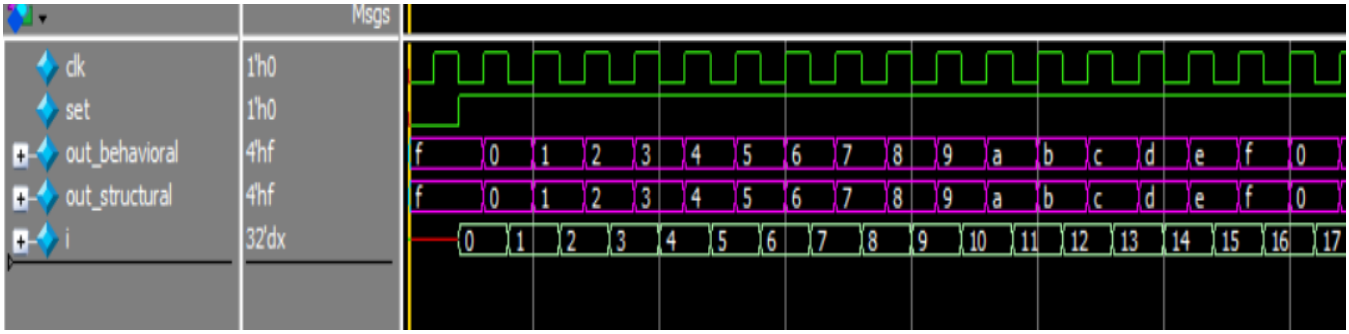
The testbench code:

```
module Counter_golden_tb();
 reg clk,set;
 wire [3:0] out_behavioral,out_structural;
 Ripple_counter golden(clk,set,out_structural);
 Counter  dut(clk,set,out_behavioral);
   initial begin
      clk=0;
      forever
      #1 clk=~clk;
      end
 integer i;
   initial begin
      set=0;
      @(negedge clk);
      if(out_behavioral!=out_structural)begin
         $display("Errrrorr");
         $stop;
      end
      set=1;
      for(i=0;i<50;i=i+1)begin
         @(negedge clk);
         if(out_behavioral!=out_structural)begin
         $display("Errrrorr");
         $stop;
```

```
        end
      end
      $stop;
    end
endmodule
```

The do file code:

```
vlib work

vlog D_Flipflop.v Ripple_counter.v Counte.v Counte_tb.v

vsim -voptargs=+acc Counter_golden_tb

add wave *

run -all

#quit -sim
```



[Q4]

4) Extend on the previous counter done in question 3 to have extra 2 single bit outputs (div_2 and div_4). Hint: Observe the output bits of the "out" bus to generate the following
   - div_2: output signal that acts as output clock with a frequency half the input clock signal.
   - div_4: output signal that acts as output clock with a frequency quarter the input clock signal.

The design code:

```
module  Extend_counter(clk,set,out,div_2,div_4);
 input clk,set;
 output reg [3:0] out ;
 output div_2,div_4;
   always @(posedge clk or negedge set) begin
      if(~set)
         out<=4'b1111;
      else
         out<=out+1;
      end
 assign div_2 = out[0];
 assign div_4 = out[1] ;
endmodule
```

The testbench code:

```
module  Extend_counter_tb();
 reg clk,set;
 wire [3:0] out_extend,out_structural;
 wire div_2_extend,div_4_extend,div_2_structural,div_4_structural;
 Ripple_counter golden(clk,set,out_structural);
 Extend_counter  dut(clk,set,out_extend,div_2_extend,div_4_extend);
 assign div_2_structural = out_structural[0];
 assign div_4_structural = out_structural[1] ;
   initial begin
      clk=0;
      forever
         #1 clk=~clk;
   end
 integer i;
   initial begin
      set=0;
      @(negedge clk);
      if(out_extend!=out_structural) $display("Errrrorr in output");
      if(div_2_extend!=div_2_structural) $display("Errrrorr in div_2");
      if(div_4_extend!=div_4_structural) $display("Errrrorr in div_4");
      set=1;
```

```
    for (i=0;i<50;i=i+1) begin
      @(negedge clk);
      if(out_extend!=out_structural) $display("Errrrorr in output");
      if(div_2_extend!=div_2_structural) $display("Errrrorr in div_2");
      if(div_4_extend!=div_4_structural) $display("Errrrorr in div_4");
    end
    $stop;
  end
endmodule
```